Lab Manual

for

Linear Algebra

by

Jim Hefferon

Cover: my Chocolate Lab, Suzy.

Contents

Python and Sage

To work through the Linear Algebra in this manual you need to be acquainted with how to run Sage. Sage uses the computer language Python so we'll start with that.

Python

Python is a popular computer language, often used for scripting, that is appealing for its simple style and powerful libraries. The significance for us of 'scripting' is that Sage uses it in this way, as a glue to bring together separate parts.

Python is Free. If your operating system doesn't provide it then go to the home page www.python.org and follow the download and installation instructions. Also at that site is Python's excellent tutorial. That tutorial is thorough; here you will see only enough Python to get started.

Comment. There is a new version, Python 3, with some differences. Here we stick to the older version because that is what Sage uses. The examples below are produced directly from Python and Sage when this manual was generated, so they should be exactly what you see, unless your version is quite different than mine. Here is my version of Python.

```
>>> import sys
>>> print sys.version
2.7.3 (default, Aug  1 2012, 05:16:07)
[GCC 4.6.3]
```

Basics Start Python, for instance by entering `python` at a command line. You'll get a couple of lines of information followed by three greater-than characters.

```
>>>
```

This is a prompt. It lets you experiment: if you type Python code and ⟨*Enter*⟩ then the system will read your code, evaluate it, and print the result. We will see below how to write and run whole programs but for now we will experiment. You can always leave the prompt with ⟨*Ctrl*⟩-*D*.

Try entering these expressions (double star is exponentiation).

```
>>> 2 - (-1)
3
>>> 1 + 2*3
7
>>> 2**3
8
```

Part of Python's appeal is that doing simple things tend to be easy. Here is how you print something to the screen.

```
>>> print 1, "plus", 2, "equals", 3
1 plus 2 equals 3
```

Often you can debug just by putting in commands to print things, and having a straightforward print operator helps with that.

As in any other computer language, variables give you a place to keep values. The first line below puts one in the place called `i` and the second line uses that.

```
>>> i = 1
>>> i + 1
2
```

In some programming languages you must declare the 'type' of a variable before you use it; for instance you would have to declare that i is an integer before you could set $i = 1$. In contrast, Python deduces the type of a variable based on what you do to it — above we assigned 1 to i so Python figured that it must be an integer. Further, we can change how we use the variable and Python will go along; here we change what is in x from an integer to a string.

```
>>> x = 1
>>> x
1
>>> x = 'a'
>>> x
'a'
```

Python complains by *raising an exeception* and giving an error message. For instance, we cannot combine a string and an integer.

```
>>> 'a'+1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

The error message's bottom line is the useful one.

Make a comment of the rest of the line with a hash mark `#`.

```
>>> t = 2.2
>>> d = (0.5) * 9.8 * (t**2)  # d in meters
>>> d
23.716000000000005
```

(Comments are more useful in a program than at the prompt.) Programmers often comment an entire line by starting that line with a hash.

As in the listing above, we can get real numbers and even complex numbers.[1]

[1] Of course these aren't actually real numbers, instead they are floating point numbers, a system that models the reals and is built into your computer's hardware. In the prior example the distinction leaks through since its bottom line ends in 000000000005, marking where the computer's binary approximation does not perfectly match the real that you expect. Similarly Python's integers aren't the integers that you studied in grade school since there is a largest one (give Python the command `import sys` followed by `sys.maxint`). However, while the issue of representation is fascinating — see Python Team [2012] and Goldberg [1991] — we shall ignore it and just call them integers and reals.

```
>>> 5.774 * 3
17.322
>>> (3+2j) - (1-4j)
(2+6j)
```

As engineers do, Python uses j for the square root of -1, not i as is traditional in Mathematics.

The examples above show addition, subtraction, multiplication, and exponentiation. Division has an awkward point. Python was originally designed to have the division bar `/` mean real number division when at least one of the two numbers is real. However between two integers the division bar was taken to mean a quotient, as in "2 goes into 5 with quotient 2 and remainder 1."

```
>>> 5.2 / 2.0
2.6
>>> 5.2 / 2
2.6
>>> 5 / 2
2
```

Experience shows this was a mistake. One of the changes in Python 3 is that the quotient operation has become `//` while the single-slash operator is always real division. In the Python 2 we are using, you must make sure that at least one number in a division is real.

```
>>> x = 5
>>> y = 2
>>> (1.0*x) / y
2.5
```

Incidentally, the integer remainder operation (sometimes called 'modulus') uses a percent character: `5 % 2` returns 1.

Variables can also represent truth values; these are *Booleans*.

```
>>> yankees_stink = True
>>> yankees_stink
True
```

You need the initial capital: `True` or `False`, not `true` or `false`.

Above we saw a string consisting of text between single quotes. You can use either single quotes or double quotes, as long as you use the same at both ends of the string. Here `x` and `y` are double-quoted, which makes sense because they contain apostrophes.

```
>>> x = "I'm Popeye the sailor man"
>>> y = "I yam what I yam and that's all what I yam"
>>> x + ', ' + y
"I'm Popeye the sailor man, I yam what I yam and that's all what I yam"
```

The `+` operation concatenates strings. Inside a double-quoted string use slash-n `\n` to get a newline.

A string marked by three sets of quotes can contain line breaks.

```
>>> a = """THE ROAD TO WISDOM
...
... The road to wisdom?
... -- Well, it's plain
... and simple to express:
... Err
```

```
... and err
... and err again
... but less
... and less
... and less. --Piet Hein"""
```

The three dots at the start of lines after the first is Python's read-eval-print prompt telling you that what you have typed is not complete. We'll see below that a common use for triple-quoted strings is as documentation.

A Python *dictionary* is a finite function. That is, it is a finite set of ordered pairs $\langle key, value\rangle$ subject to the restriction that no key can be used twice. Dictionaries are a simple database.

```
>>> english_words = {'one': 1, 'two': 2, 'three': 3}
>>> english_words['one']
1
>>> english_words['four'] = 4
>>> english_words
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
```

Don't be mislead by this example, the words do not always just come in the reverse of the order in which you entered them. A dictionary's elements will be listed in no apparently-sensible order.

If you assign to an existing key then that will replace the previous value.

```
>>> english_words['one'] = 5
>>> english_words
{'four': 4, 'three': 3, 'two': 2, 'one': 5}
```

Dictionaries are central to Python, in part because looking up values in a dictionary is very fast.

While dictionaries are unordered, a *list* is ordered.

```
>>> a = ['alpha', 'beta', 'gamma']
>>> b = []
>>> c = ['delta']
>>> a
['alpha', 'beta', 'gamma']
>>> a+b+c
['alpha', 'beta', 'gamma', 'delta']
```

Get an element from a list by specifying its index, its place in the list, inside square brackets. Lists are *zero-offset* indexed, that is, the initial element of the list is numbered 0. Count from the back by using negative indices.

```
>>> a[0]
'alpha'
>>> a[1]
'beta'
>>> a[-1]
'gamma'
```

Specifying two indices separated by a colon gets a *slice* of the list.

```
>>> a[1:3]
['beta', 'gamma']
>>> a[1:2]
['beta']
```

You can add to a list.

```
>>> c.append('epsilon')
>>> c
['delta', 'epsilon']
```

Lists can contain anything, including other lists.

```
>>> x = 4
>>> a = ['alpha', [True, x]]
>>> a
['alpha', [True, 4]]
```

The function `range` returns a list of numbers.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

By default `range` starts at 0, which is good because lists are zero-indexed. Observe also that 9 is the highest number in the list given by `range(10)`. This makes `range(10)+range(10,20)` give the same list as `range(20)`.

A *tuple* is like a list in that it is ordered.

```
>>> a = ('fee', 'fie', 'foe', 'fum')
>>> a
('fee', 'fie', 'foe', 'fum')
>>> a[0]
'fee'
```

However it is unlike a list in that a tuple is not *mutable*—it cannot change.

```
>>> a[0] = 'phooey'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

One reason this is useful is that because of it tuples can be keys in dictionaries while list cannot.

```
>>> a = ['Jim', 2138]
>>> b = ('Jim', 2138)
>>> d = {a: 'active'}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d = {b: 'active'}
>>> d
{('Jim', 2138): 'active'}
```

Python has a special value `None` for when there is no sensible value for a variable. For instance, if your program keeps track of a person's address and includes a variable `apartment` then `None` is the right value for that variable when the person does not live in an apartment.

Flow of control Python supports the traditional ways of affecting the order of statement execution, with a twist.

```
>>> x = 4
>>> if (x == 0):
...     y = 1
... else:
...     y = 0
...
>>> y
0
```

The twist is that while many languages use braces or some other syntax to mark a block of code, Python uses indentation. (Always indent with four spaces.) Here, Python executes the single-line block `y = 1` if x equals 0, otherwise Python sets y to 0.

Notice also that double equals == means “is equal to.” In contrast, we have already seen that single equals is the assignment operation so that `x = 4` means “x is assigned the value 4.”

Python has two variants on the above `if` statement. It could have only one branch

```
>>> x = 4
>>> y = 0
>>> if (x == 0):
...     y = 1
...
>>> y
0
```

or it could have more than two branches.

```
>>> x = 2
>>> if (x == 0):
...     y = 1
... elif (x == 1):
...     y = 0
... else:
...     y = -1
...
>>> y
-1
```

Computers excel at iteration, looping through the same steps.

```
>>> for i in range(5):
...     print i, "squared is", i**2
...
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

A `for` loop often involves a `range`.

```
>>> x = [4, 0, 3, 0]
>>> for i in range(len(x)):
...     if (x[i] == 0):
...         print "item",i,"is zero"
...     else:
```

```
...         print "item",i,"is nonzero"
...
item 0 is nonzero
item 1 is zero
item 2 is nonzero
item 3 is zero
```

An experienced Python person who was not trying to illustrate `range` would instead writte `for c in x:` since the `for` loop can iterate over any sequence, not just a sequence of integers.

A `for` loop is designed to execute a certain number of times. The natural way to write a loop that will run an uncertain number of times is `while`.

```
>>> n = 27
>>> i = 0
>>> while (n != 1):
...     if (n%2 == 0):
...         n = n / 2
...     else:
...         n = 3*n + 1
...     i = i + 1
...     print "i=", i
...
i= 1
i= 2
```

(This listing is incomplete; it takes 111 steps to finish.)[1] Note that "not equal" is `!=`.

The `break` command gets you out of a loop right away.

```
>>> for i in range(10):
...     if (i == 3):
...         break
...     print "i is", i
...
i is 0
i is 1
i is 2
```

A common loop construct is to run through a list performing an action on each entry. Python has a shortcut for this, *list comprehension*.

```
>>> a = [2**i for i in range(4)]
>>> a
[1, 2, 4, 8]
>>> [i-1 for i in a]
[0, 1, 3, 7]
```

Functions A *function* is a group of statements that executes when it is called, and can return values to the caller. Here is a naive version of the quadratic formula.

```
>>> def quad_formula(a, b, c):
...     discriminant = (b**2 - 4*a*c)**(0.5)
...     r1=(-1*b+discriminant) / (2.0*a)
```

[1]The *Collatz conjecture* is that for any starting n this loop will terminate, but this is not known.

```
...     r2=(-1*b-discriminant) / (2.0*a)
...     return (r1, r2)
...
>>> quad_formula(1,0,-9)
(3.0, -3.0)
>>> quad_formula(1,2,1)
(-1.0, -1.0)
```

(One way that it is naive is that it doesn't handle complex roots gracefully.) Functions organize code into blocks that may be run a number of different times or which may belong together conceptually. In a Python program the great majority of code is in functions.

At the end of the `def` line, in parentheses, are the function's *parameters*. These can take values passed in by the caller. Functions can have *optional parameters* that have a default value.

```
>>> def hello(name="Jim"):
...     print "Hello,", name
...
>>> hello("Fred")
Hello, Fred
>>> hello()
Hello, Jim
```

Sage uses this aspect of Python a great deal.

Functions can contain multiple `return` statements. They always return something; if a function never executes a `return` then it will return the value `None`.

Objects and modules In Mathematics, the real numbers is a set associated with some operations such as addition and multiplication. Python is *object-oriented*, which means that we can similarly bundle together data and actions.

```
>>> class person(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def hello(self):
...         print "Hello", self.name
...
>>> a=person("Jim", 53)
>>> a.hello()
Hello Jim
>>> a.age
53
```

You work with objects by using the *dot notation*: to get the age data bundled with `a` you write `a.age`, and to call the `hello` function bundled with `a` you write `a.hello()` (a function bundled in this way is called a *method*).

You won't be writing your own classes in this lab manual but you will be using ones from the extensive libraries of code that others have written, including the code for Sage. For instance, Python has a library, or *module*, for math.

```
>>> import math
>>> math.pi
3.141592653589793
```

```
>>> math.factorial(4)
24
>>> math.cos(math.pi)
-1.0
```

The `import` statement gets the module and makes its contents available.

Another library is for random numbers.

```
>>> import random
>>> while (random.randint(1,10) != 1):
...     print "wrong"
...
wrong
wrong
```

Programs The read-eval-print loop is great for small experiments but for more than four or five lines you want to put your work in a separate file and run it as a standalone program.

To write the code, use a text editor; one example is Emacs[1]. You should try to use an editor with support for Python such as automatic indentation, and syntax highlighting, where the editor colors your code to make it easier to read.

Here is one example. Start your editor, open a file called `test.py`, and enter these lines. Note the triple-quoted documentation string at the top of the file; good practice is to include this documentation in everything you write.

```
# test.py
"""test

A test program for Python.
"""

import datetime

current = datetime.datetime.now()  # get a datetime object
print "the month number is", current.month
```

Run it under Python (for instance, from the command line run `python test.py`) and you should see output like `the month number is 9`.

Here is a small game (it has some Python constructs that you haven't seen but that are straightforward).

```
# guessing_game.py
"""guessing_game

A toy game for demonstration.
"""
import random
CHOICE = random.randint(1,10)

def test_guess(guess):
    """Decide if the guess is correct and print a message.
```

[1]It may come with your operating system or see http://www.gnu.org/software/emacs.

```
    """
    if (guess < CHOICE):
        print "  Sorry, your guess is too low"
        return False
    elif (guess > CHOICE):
        print "  Sorry, your guess is too high"
        return False
    print "  You are right!"
    return True

flag = False
while (not flag):
    guess = int(raw_input("Guess an integer between 1 and 10: "))
    flag = test_guess(guess)
```

Here is the output.

```
$ python guessing_game.py
Guess an integer between 1 and 10: 5
  Sorry, your guess is too low
Guess an integer between 1 and 10: 8
  Sorry, your guess is too high
Guess an integer between 1 and 10: 6
  Sorry, your guess is too low
Guess an integer between 1 and 10: 7
  You are right!
```

As above, note the triple-quoted documentation strings both for the file as a whole and for the function. Go to the directory containing `guessing_game.py` and start Python. At the prompt type in `import guessing_game`. You will play through a round of the game (there is a way to avoid this but it doesn't matter here) and then type `help("guessing_game")`. You will see documentation that includes these lines.

```
DESCRIPTION
    A toy game for demonstration.

FUNCTIONS
    test_guess(guess)
        Decide if the guess is correct and print a message.
```

Obviously, Python got this from the file's documentation strings. In Python, and also in Sage, good practice is to always include documentation that is accessible with the `help` command.

Sage

Learning what Sage can do is the object of much of this book so this is a very brief walk-through of preliminaries. See [Sage Development Team, 2012] for a more broad-based introduction.

First, if your system does not already supply it then install Sage by following the directions at www.sagemath.org.

Command line Sage's command line is like Python's but adapted to mathematical work. First start Sage, for instance, enter `sage` into a command line window. You get some initial text and then a prompt (leave the prompt by typing `exit` and ⟨*Enter*⟩.)

```
sage:
```

Experiment with some expressions.

```
sage: 2**3
8
sage: 2^3
8
sage: 3*1 + 4*2
11
sage: 5 == 3+3
False
sage: sin(pi/3)
1/2*sqrt(3)
```

The second expression shows that Sage provides a convenient shortcut for exponentiation. The fourth shows that Sage sometimes returns exact results rather than an approximation. You can still get the approximation.

```
sage: sin(pi/3).numerical_approx()
0.866025403784439
sage: sin(pi/3).n()
0.866025403784439
```

The function `n()` is an abbreviation for `numerical_approx()`.

Script You can group Sage commands together in a file. This way you can test the commands, and also reuse them without having to retype.

Create a file with the extension `.sage`, such as `sage_try.sage`. Enter this function and save the file.

```
def normal_curve(upper_limit):
    """Approximate area under the Normal curve from 0 to upper_limit.
    """
    stdev = 1.0
    mu = 0.0
    area=numerical_integral((1/sqrt(2*pi) * e^(-0.5*(x)^2)),
                            0, upper_limit)
    print "area is", area[0]
```

Bring in the commands in with a `load` command.

```
sage: load "sage_try.sage"
sage: normal_curve(1.0)
area is 0.341344746069
```

Notebook Sage also offers a browser-based interface, where you can set up worksheets to run alone or with other people, where you can easily view plots integrated with the text, and many other nice features.

From the Sage prompt run `notebook()` and work through the tutorial.

Bibliography

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

Python Team. Floating point arithmetic: issues and limitations, 2012. URL http://docs.python.org/2/tutorial/floatingpoint.html. [Online; accessed 17-Dec-2012].

Sage Development Team. Sage tutorial 5.3. http://www.sagemath.org/pdf/SageTutorial.pdf, 2012.